Information and Computation **142**, 26–39 (1998)
Article No. IC972693

Repeated Redundant Inequalities in Constraint Logic Programming*

Spiro Michaylov†

Department of Computer and Information Science, The Ohio State University, 395 Dreese Laboratory, 2015 Neil Avenue, Columbus, Ohio 43210-1277
E-mail: Spiro.Michaylov@East.Sun.Com

Future redundant inequalities in a constraint logic program are those that are guaranteed to be subsumed after no more than one subsequent procedure call, whenever they are encountered at run time. It has been noted that such inequalities need only be tested for consistency with the current constraint set, thus resulting in dramatic savings in execution speed and space usage. Furthermore, they can be detected at compile time, leading to a valuable compiler optimization. By considering the role of such inequalities in a program, we generalize the notion of future redundancy in a number of ways and thus broaden its applicability. As a result, we show how to dramatically improve the performance of a wider class of programs that rely heavily on inequalities.

1. INTRODUCTION

A number of constraint logic programming (CLP) systems (Jaffar and Lassez, 1987), including CLP($\mathscr{R}$) and Prolog III, decide simultaneous linear inequalities as part of the fundamental operational step of constraint solving. While this contributes to the usefulness of the systems, it is computationally expensive. Non-ground inequalities must be tested for consistency with the collected constraint set and then added to it, increasing its size, hence making the next such test more expensive.

We will discuss this problem in the context of CLP($\mathscr{R}$) (Jaffar *et al.*, 1992b), a CLP language dealing with real arithmetic, although the results apply to other languages that decide inequalities. A locally optimizing compiler for CLP($\mathscr{R}$), utilizing an abstract code interpreter, was described in (Jaffar *et al.*, 1992a). Global optimization techniques for CLP($\mathscr{R}$) and related languages have been discussed by (Jørgensen *et al.*, 1991; Marriott and Søndergaard, 1990; Marriott and Stuckey, 1993; McDonald *et al.*, 1993; Michaylov and Pippin, 1994). For an overview, see (Michaylov, 1992). Some of the techniques, such as the future redundancy

* This research is supported in part by the National Science Foundation Grant #CCR-9308631.
† Current address: Sun Microsystems, Inc., 5 Omni Way, Chelmsford, MA 01824.

0890-5401/98 $25.00

optimization in (Jørgensen *et al.*, 1991), also rely on analyzing the patterns of constraint invocation for classes of queries, and observing how the constraints interact with each other.

Future redundancy of arithmetic inequality constraints was first defined in (Jørgensen *et al.*, 1991). Future redundant inequalities in a program are those that are guaranteed to be subsumed after no more than one subsequent procedure call, whenever they are encountered at run time. While inequalities must normally be tested for consistency with the collected constraint set and then added to it, future redundant inequalities need only be tested for consistency, thus resulting in dramatic savings in execution speed and space usage. This optimization is closely related to the run-time constraint removal technique of Marriott and Stuckey (Marriott and Stuckey, 1993). As discussed in Section 6, run-time constraint removal is one good way to implement the optimizations described here. In that paper, Marriott and Stuckey show examples where programs can be optimized by removing constraints from the store once they have been made redundant in some way. However, they do not discuss how to detect any specific case of such redundancy except where the subsumption occurs within the body of a single rule. In general it would be difficult to directly extend their method to future redundancy.

In this paper, we consider the essential features of individual procedures (rules defining one relation) that lead to the repeated uniform subsumption of linear inequalities. Thus we define a class of procedures that lead to such subsumption, show that this class is important in practice, and consider how it can be optimized. We show that the future redundancy optimization due to Jørgensen *et al.* is not sufficiently general to optimize these, and that more general notions of future redundancy are needed. We then describe appropriate optimizations in the form of source-to-source transformations. Our contribution may be viewed in two ways: from the viewpoint of programming methodology and language implementation. We identify a potentially important programming technique as being "safe" in the sense that the resulting programs do not suffer from the inefficiency that had been expected, and we identify a procedure-level optimizing transformation for programs that can result in dramatic performance improvement, having developed a good understanding of why and how the transformation is applicable in real programs.

This work is heavily influenced by the highly structural view of recursive procedures that is used for query optimization in deductive databases. The basic approach is similar to that used when queries or intensional databases contain constraints, as in (Kemp *et al.*, 1989; Kemp and Stuckey, 1993; Stuckey and Sudarshan, 1994). However, there the propagation of values or constraints from queries tends to be central to the optimizations, and queries are not merely used to choose between specializations. The core of our work is the detection of relationships among constraints in a procedure, although properties of queries are used to expose relationships that may not hold in general. Thus, the emphasis and specific properties detected are different, but the basic approach and philosophy are similar.

The remainder of this paper is organized as follows. We examine the connection between static program structure, calling patterns, and the run-time subsumption of

linear inequalities in Section 2. In Section 3, we make some preliminary definitions, redefine the original notion of future redundancy in our context, and restate the corresponding optimization and its correctness. Then in Section 4 we introduce the notion of partial future redundancy, and its corresponding optimization. Return future redundancy is introduced in the same manner in Section 5. Some pragmatic issues including multiple future redundancy and multiple specialization are briefly discussed in Section 6. Section 7 gives details of performance improvement for a number of examples.

2. PROGRAMS, QUERIES, AND REDUNDANT INEQUALITIES

We begin by developing an understanding of a class of programs and queries that lead to inequalities being systematically subsumed at run time. Consider the following loop, called with the second argument free.

```
p(X, Y, Z):-
        X>1, NY=Y+20, NY<10000, p(X-1, NY, Z).
```

Each of the inequalities on `NY` will be subsumed by the following one in conjunction with intervening equations. That is,

$$NY' < 10000 \wedge NY' = Y + 20 \wedge Y = NY \Rightarrow NY < 10000,$$

where we use NY' for the second instance of NY. Thus we would like to simply test the consistency of the inequality $NY < 10000$ with the solver state, without *adding* it to the solver state. We must check, however, that doing so could not affect program behavior. A recursive call always follows the constraint immediately. If the recursive call uses the same rule, the constraint is immediately subsumed. However, if there are other rules for `p/3`, the computation could proceed erroneously. Thus, we require that *each* of the other rules for `p/3` also has, before the first subsequent call, a conjunction of constraints that subsumes `NY<10000`. This was defined as the *future redundancy* condition of (Jørgensen *et al.*, 1991). It requires, in particular, that every base case also results in the constraint being subsumed.

The future redundancy condition captures correctly an important class of constraints that can be tested without being added to the constraint solver. However, by examining the class of linear recursive procedures we see that there are other important cases of constraints being systematically subsumed, perhaps with exceptions at boundary conditions, that are not captured by this definition.

Consider constraints that will be subsumed by a call to any recursive rule, but not to some base cases. We will call these partially future redundant. A successful,

```
p1(0, W, W).
p1(X, Y, Z) :-
        X > 0, NY = Y + 20, NY < 10000, p1(X - 1, NY, Z).
```

FIG. 1. Summation loop that is partially future redundant.

```
p2(0, W, W).
p2(X, NY, Z) :-
        X > 0, p2(X - 1, Y, Z), NY = Y - 20, NY < 10000.
```

FIG. 2. Summation loop that is return future redundant.

finite computation using a linear recursive procedure makes use of exactly one of the base cases of that procedure exactly once. This means that at most one instance of any partially future redundant constraint was tested when indeed it should have been added to the constraint set. This was the last instance before the use of the base case. All other instances of the constraint actually *were* subsumed, so it was appropriate for them to be tested without being added. Consider the procedure in Fig. 1. If called with the first argument ground and second argument free, every use of the recursive rule will establish an inequality constraint that will be subsumed by the following use of that rule, except for the last use before the base case. Thus, for example, the query `?-p1(10,U,V)` should result in the answer constraint $V = U + 200 \wedge V < 10000$ as the last inequality must be added to the solver. As we shall see in Section 4, this can be achieved by a general source to source transformation, even for more general cases where different recursive rules establish different inequalities, and some of the base cases subsume these inequalities while others do not.

Now consider the following procedure `p2` in Fig. 2. When called with the second argument free and third argument ground, a chain of inequalities is established on NY. Each of these is subsumed immediately after the return from the body of the rule, except for that in the outermost use of the rule. We call the inequality on NY return future redundant, and the associated optimization is described in Section 5.

This example is chosen for simplicity and may seem contrived. In fact it may seem that any such program could be rewritten (perhaps by the compiler) so the constraints precede the recursive call. In general this is not practical, because the "bottoming out" of the recursion may be needed to ground certain variables, which in turn may be needed to make other constraints linear. Reordering the constraints could introduce delaying of nonlinear constraints making the program even less efficient—especially if it causes redundant inequalities to be nonlinear.

3. FUTURE REDUNDANCY

Here we define future redundancy in a somewhat different manner from that in (Jørgensen *et al.*, 1991). It is more general in the sense that the redundant constraint need not appear before the first atom in the body. On the other hand, it is restricted to linear recursive procedures, on the assumption that recursive "loops" are the main source of inequality subsumption. We assume all programs to be in canonical form, in the sense that only variables appear as arguments of predicates in the body or head. We refer to head arguments as a vector $\tilde{X}$ of variables and to those of a recursive call to the procedure as the vector $\tilde{Y}$. We use $\tilde{C}, \tilde{D}, \tilde{E}, \ldots$ for sequences of constraints and atoms in the body of a rule, ρ for a rule, and π for

individual constraints. We also assume the usual left-right atom and constraint selection rule. Now we define the broad class of procedures that we wish to optimize.

DEFINITION 1 (Linear Recursive Procedure). A linear recursive procedure P is a sequence of rules of the form $p(\tilde{X}) :- \tilde{G}$, where $\tilde{G}$ is a possibly empty sequence of atoms and constraints. Each rule is either

1. *A Base case*: None of the atoms in $\tilde{G}$ may result in a recursive call to P.
2. *A recursive Case*: Exactly one of the atoms in $\tilde{G}$ is a recursive call to P, but no other atom in $\tilde{G}$ may result in such a call indirectly.

We begin with a minor definition of the constraints that result from a single procedure call and the immediately surrounding constraints in both the calling and called rules.

DEFINITION 2 (Derived Constraint). Let A be an atom and ρ be the rule $H: -\tilde{C}, \tilde{D}$, where $\tilde{C}$ is a sequence of constraints and $\tilde{D}$ is a sequence of atoms and constraints. Then $Der(A, \rho) \equiv (A = H) \wedge \tilde{C}$ is their *derived constraint*: the result of reducing subgoal A using rule ρ.

DEFINITION 3 (Future Redundancy). Let

$$p(\tilde{X}) :- \tilde{C}, \pi, \tilde{D}, p(\tilde{Y}), \tilde{E}$$

be a rule in a linear recursive procedure P, where $\tilde{C}$ and $\tilde{E}$ are sequences of constraints and atoms, π is a constraint, and $\tilde{D}$ is a sequence of constraints. Constraint π is *future redundant* in that rule if, for every rule $\rho \in P$,

$$Der(p(\tilde{Y}), \rho) \wedge \tilde{D} \Rightarrow \pi.$$

Next, we show how to systematically transform such a program so that the collected set of inequality constraints does not grow with each iteration. We begin by extending the operational model to cope with two meta-level predicates on constraints: `test/1` and `add/1`. The former checks that a constraint is consistent with the collected constraint set, but does not add it to that set. The latter assumes that it is consistent and adds it to the collected constraint set. In the following definition, we use C?-$\tilde{G}$ to denote that the sequence $\tilde{G}$ of constraints and atoms is to be solved in the context of the constraint store C. We briefly discuss the implementation of the `test/1` predicate in Section 6.

DEFINITION 4 (Execution of `test/1` and `add/1`). The CLP operational model is augmented with the following two transitions:

1. The goal C?-test(π), $\tilde{G}$ reduces to the goal C?-$\tilde{G}$ if $C \cup \{\pi\}$ is satisfiable.
2. The goal C?-add(π), $\tilde{G}$ reduces to the goal $C \cup \{\pi\}$?-$\tilde{G}$.

TRANSFORMATION 1 (Future Redundancy Optimization). *For any linear recursive procedure, replace each future redundant constraint π with the atom test(π).*

The correctness of this transformation may then be stated as in (Jørgensen *et al.*, 1991).

PROPOSITION 1 (Correctness of Future Redundancy Optimization). *There is a one-to-one correspondence between* (*possibly partial*) *derivations for any program and that program after the future redundancy transformation, in which derivation length is preserved* (*to within one resolution step for unsuccessful derivations*). *Furthermore, in the case of successful derivations, the answer constraints are equivalent, modulo variable renaming.*

Proof of Proposition 1. By induction on derivation length,

- *Base case*: This corresponds to a base case of the procedure, and these are unchanged.
- *Inductive case*: This corresponds to using a recursive rule. If there is no future redundant constraint in this rule, there is nothing to check. Otherwise consider future redundant constraint π in rule $p(\tilde{X}) :- \tilde{C}, \pi, \tilde{D}, p(\tilde{Y}), \tilde{E}$ and an attempt to then resolve $p(\tilde{Y})$ with rule ρ:

— $\pi \wedge \tilde{D} \wedge Der(p(\tilde{Y}), \rho)$ is consistent with the store: then in both programs the resolution step will succeed.

— $\pi \wedge \tilde{D} \wedge Der(p(\tilde{Y}), \rho)$ is not consistent with the store: since π is future redundant, $\tilde{D} \wedge Der(p(\tilde{Y}), \rho) \Rightarrow \pi$, so $\tilde{D} \wedge Der(p(\tilde{Y}), \rho)$ is not consistent with the store.

∗ π is not consistent with the store: the resolution step fails in both programs.

∗ π is consistent with the store: the resolution step may succeed, but since $\tilde{D} \wedge Der(p(\tilde{Y}), \rho)$ is not consistent with the store, there will then be a failure before the next resolution step.

Answer equivalence is maintained since any constraint not added to the store is subsumed by some constraint encountered later, and such chains of subsumptions are always terminated by a rule without future redundant constraints. ▌

In (Jørgensen *et al.*, 1991), the notion of future redundancy with respect to a given calling pattern is also defined. The definitions in this paper can be modified similarly.

4. PARTIAL FUTURE REDUNDANCY

Now we give our generalized definition.

DEFINITION 5 (Partial Future Redundancy). Let

$$p(\tilde{X}) :- \tilde{C}, \pi, \tilde{D}, p(\tilde{Y}), \tilde{E}$$

be a rule in a linear recursive procedure P, where $\tilde{C}$ and $\tilde{E}$ are sequences of constraints and atoms, π is a constraint, and $\tilde{D}$ is a sequence of constraints. Constraint

π is *partially future redundant* in that rule if, for every rule $\rho \in P$ that is a recursive case,

$$Der(p(\tilde{Y}), \rho) \wedge \tilde{D} \Rightarrow \pi.$$

This definition differs only subtly from that of future redundancy: ρ now ranges over all recursive cases in the procedure, rather than all rules. However, the transformation required differs quite substantially.

TRANSFORMATION 2 (Partial Future Redundancy Optimization). *A linear recursive procedure p/n containing partial future redundant constraints becomes $p/(n+1)$, where the extra argument is used for passing administrative information. We assume the existence of a set of unique constants not already present in the program, denoted by variously subscripted c.*

1. *Add as a wrapper the new rule* $p(\tilde{X}) :- p(\tilde{X}, c_0)$.

2. *For every base case: add as the* $(n+1)$*th head argument the term* c_0. *Call the resulting rules the original base cases.*

3. *For every recursive rule* ρ:

   (a) *Add as* $(n+1)$*th head argument the anonymous variable* (_).

   (b) *For the recursive call*:

   (i) *If there is no associated partially future redundant constraint, add* c_0 *as the* $(n+1)$*th argument. Otherwise, add as the* $(n+1)$*th argument the term* $c_\rho(\tilde{V}_\rho)$, *where* $\tilde{V}_\rho$ *consists of the variables in the partially future redundant constraint corresponding to that call.*

   (ii) *If there is a partially future redundant constraint* π_ρ, *replace it with test*(π_ρ), *and add, immediately after each original base case, a new version with* $(n+1)$*th head argument* $c_\rho(\tilde{W})$ *where* $\tilde{W}$ *is a unique renaming of* $\tilde{V}_\rho$, *and with added subgoal add*$(\pi_{\tilde{W}})$ *before any atoms in the rule.*

If the original procedure had r recursive rules, with a total of s recursive calls with associated partially future redundant constraints, and b base cases, the transformed procedure has one wrapping rule, r recursive rules, and $b(s+1)$ base cases. However, only b base cases have any chance of unifying with any given call to the procedure, because of the effect of the $(n+1)$th argument. Hence, some relatively minor local optimizations such as indexing can be used to ensure that the overheads of the transformed program are not excessive, provided that more than a few iterations will be taken.

Note that a slightly simpler transformation could be defined if the final inequality was to be added after returning from the procedure. However, the present optimization preserves more of the operational behavior of the original program.

PROPOSITION 2 (Correctness of Partial Future Redundancy Optimization). *There is a one-to-one correspondence between (possibly partial) derivations for any program and that program after the partial future redundancy transformation, in which derivation length is preserved (to within one resolution step for unsuccessful*

```
p2(X, Y, Z) :- p2(X, Y, Z, a).

p2(0, W, W, a).
p2(0, W, W, b(X)) :-
        add(X < 10000).
p2(X, Y, Z, _) :-
        X > 0, NY = Y + 20, test(NY < 10000), p2(X - 1, NY, Z, b(NY)).
```

FIG. 3. Optimizing the partially future redundant summation loop from Fig. 1.

derivations). *Furthermore, in the case of successful derivations, the answer constraints are equivalent, modulo variable renaming.*

Proof of Proposition 2. By induction on derivation length much as in Proposition 1. The induction is unrolled by one step because this time the base cases in the transformed program may need to add constraints to "clean up" after the last recursive rule.

- Using a base case rule:

— No recursive rules were used in the original derivation: The original base case is used, adding no extra constraints, as guaranteed by the extra argument in the call from the wrapper.

— At least one recursive rule was used in the original derivation: If the last recursive rule used contained a partially future redundant constraint, the extra argument to the recursive call in the transformed rule ensures that the appropriate base case is used to add the constraints. If the base case causes failure, the failed derivation was lengthened by one step.

- Using a recursive rule:

— The next resolution step is with a base case: covered above.

— The next resolution step is with another recursive rule: Similar to the inductive case in the proof of Proposition 1.

Answer equivalence is maintained by the base case as before, unless the original base case is used—then an earlier recursive rule must have provided the subsumption. ▮

In Fig. 3 we show the effect of this transformation on a simple example. The optimized code has been simplified to remove some redundancy.

5. RETURN FUTURE REDUNDANCY

We now turn our attention to those linear recursive procedures where the inequality is located after the recursive call. Such inequalities can be considered, in a sense, future redundant if they are subsumed immediately after the return from a call in which they are encountered. Now of course this situation is analogous to

partial future redundancy, since the last instance of the constraint will most likely *not* be subsumed.

In the following we define a notion of derivation step corresponding to that of having collected additional constraints after returning from a procedure call, but ignoring one constraint in the call (π), which corresponds to the redundant constraint. Note that all other constraints of the called and calling rule are used.

DEFINITION 6 (Return-Derived Constraint). Let ρ_π and ρ be two of the rules defining a linear recursive procedure P, not necessarily distinct, such that:

1. ρ_π contains a constraint π in its body, has head arguments $\tilde{Y}$, and $\tilde{C}_\pi$ is the sequence of all its body constraints other than π;

2. ρ has arguments $\tilde{X}$ in its recursive call, and $\tilde{C}$ is the sequence of all its body constraints.

Then, the return-derived constraint resulting from resolving the recursive call in ρ with the rule ρ_π with respect to the constraint π is: $Ret(\pi, \rho_\pi, \rho) = \tilde{C}_\pi \wedge (\tilde{X} = \tilde{Y}) \wedge \tilde{C}$.

Now using this we may define what it means for a constraint to be made redundant immediately after a call to the rule containing it terminates.

DEFINITION 7 (Return Future Redundancy). Let P be a linear recursive procedure. The constraint π is *return future redundant* in the recursive rule ρ_π of P if it appears after the last atom, and in each recursive rule ρ in P, no atoms follow the recursive call, and $Ret(\pi, \rho_\pi, \rho) \Rightarrow \pi$.

It is possible to define a more general version where atoms are allowed after the recursive call, but the conditions are quite restrictive and the details would obscure the presentation.

We may now define a corresponding optimization for linear recursive procedures containing return future redundant constraints. Note that in these definitions we ignore the possibility of return future redundant constraints in base cases, as they do not lead to important optimizations.

TRANSFORMATION 3 (Return Future Redundancy Optimization). *A linear recursive procedure p/n containing return future redundant constraints becomes a wrapper defining p/n; a procedure $p/(n+1)$, where the extra argument is used for passing administrative information; and a tidy-up procedure addlast/1, defined as follows:*

1. *Add as a wrapper the new rule* $p(\tilde{X}) : - p(\tilde{X}, Z), addlast(Z)$.

2. *For every base case: add as the* $(n+1)$*th head argument the term* c_0.

3. *Add a rule* $addlast(c_0)$ *to handle a case where only a base case was used and hence no return future redundant constraints were encountered.*

4. *For every recursive rule* ρ:

(a) *Add as* $(n+1)$*th head argument the term* $c_\rho(\tilde{V}_\rho)$, *where* $\tilde{V}_\rho$ *consists of the variables in the return future redundant constraint in the body.*

(b) *Add as* $(n+1)$*th argument to the recursive call the anonymous variable* (_).

(c) *Replace the return future redundant constraint* π_ρ *by* $test(\pi_\rho)$.

(d) *Add the rule* $addlast(c_\rho(\tilde{V}_\rho)) : -add(\pi_\rho)$.

This optimization does not cause as much code explosion as the previous one, and the same local optimizations as before are still applicable. The case where some of the recursive rules do not contain return future redundant constraints is not mentioned explicitly, but can trivially be handled as before. In Fig. 4 we show how this algorithm leads to an optimized version of the summation loop in Fig. 2.

PROPOSITION 3 (Correctness of Return Future Redundancy Optimization). *There is a one-to-one correspondence between* (*possibly partial*) *derivations for any program and that program after the return future redundancy transformation, in which derivation length is preserved with one additional derivation for the wrapper. Furthermore, in the case of successful derivations, the answer constraints are equivalent, modulo variable renaming.*

Proof of Proposition 3. This is simpler than the previous proofs because the constraints do not affect resolution steps: they are selected only after the base case has been reached. The induction is still on the length of a derivation. The key is to realize that in the context of a left-right atom and constraint selection rule constraints after the recursive call will remain as residual constraints after the resolution step and will be selected once the base case has been reached.

- *Base case*: There is no redundant constraint, and the extra argument ensures that the extra L constraint is not added by the wrapper.
- *Inductive case*: If this is the first resolution step, the appropriate constraint will be added by the wrapper in the final resolution step. Otherwise, it is in the residual constraints from the previous resolution step and will be selected after the base case is reached.

Answer equivalence is guaranteed as before by the chain of subsumptions. ▮

Of course, both partial future redundant and return future redundant constraints may occur in one linear recursive procedure. In that case it is easy to see that the two optimizations may be combined.

```
p2(X, Y, Z) :- p2(X, Y, Z, U), addlast(U).

p2(0, W, W, a).
p2(X, NY, Z, b(NY)) :-
        X > 0, p2(X - 1, Y, Z, _), NY = Y - 20, test(NY < 10000).

addlast(a).
addlast(b(Y)) :- Y < 10000.
```

FIG. 4. Optimizing the return future redundant summation loop from Fig. 2.

6. SOME PRAGMATIC ISSUES

So far we have only considered cases of future redundancy where there is at most one redundant constraint associated with each recursive call in a rule. However, it is possible for a rule to contain multiple redundant constraints associated with one recursive call. In this case we must check the *mutual* satisfiability of the redundant constraints: the first constraint has not yet been subsumed when we test the second. This can be handled by extending the *test*/1 predicate to take conjunctions of constraints.

The issue of identifying future redundant constraints in programs at compile time was addressed by Jørgensen (Jørgensen, 1992) in the implementation of his global analyzer. Essentially, in addition to some form of abstract interpretations to narrow ranges of variables, constraints in bodies of rules need to be considered with respect to the bodies of rules that may match immediately following calls. The core operation is then a subsumption test on small sets of constraints, which is, of course, time consuming. The optimizations described here pose essentially the same problems, but the search is narrowed somewhat. Only constraints that almost immediately precede recursive calls to the same procedure need be considered, and then only with respect to matching recursive rules. Furthermore, a wide range of heuristics can be applied more easily to our versions of future redundancy, based on the structure of typical recursive procedures, since procedures are considered individually.

The `test` predicate can be implemented in a number of ways, and of course this can affect the efficiency of the optimized code. For the empirical study described below we chose a simple and rather inefficient solution that is well known in the folklore of Prolog programming:

```
test(C) :- not(not(C))
```

made slightly more efficient by (manually) partially evaluating the standard implementation of `not` with respect to the constraint at hand, since `not` is handled very poorly in CLP($\mathscr{R}$). The dramatic speedups obtained demonstrate that the inefficiency of `test` was not of great importance in these examples. However, `test` can be seen as a special case of constraint removal predicate `rem` discussed by Marriott and Stuckey (Marriott and Stuckey, 1993). We note that `test` could be implemented in terms of their removal predicate as:

```
test(C) :- C,rem(C)
```

which would be more efficient. However, it would be even better to modify the abstract machine for executing CLP($\mathscr{R}$) to test a constraint without adding it at all (although, in fact, the best way to do this using the Simplex algorithm is to add the constraint, check satisfiability, and then remove it).

Finally, we note that some other global optimizations, possibly based on multiple specialization (most fully described in (Winsborough, 1989)), subgoal reordering, or even some unfolding of simple calls may be helpful in exposing some cases of future redundancy. Conversely, the desire to expose such opportunities may

motivate the application of some of the other optimizations. In short, the practical use of optimizations such as those described here can be expected to be the subject of ongoing research for some time to come.

7. EMPIRICS

To demonstrate the usefulness of the optimizations we give time and space improvements for a range of CLP($\mathscr{R}$) loops. Since these optimizations are essentially loop transformations, we have measured the effect on actual individual loops with future redundancy rather than entire programs. The effect on real programs would of course depend on the occurrence of such loops. We have argued that they make sense in a wide range of applications and show them to be relatively innocuous from the viewpoint of performance.

These results were obtained on a SPARC ELC with 64 MB of memory, using an instrumented variant of Version 1.1 of IBM's CLP($\mathscr{R}$) compiler. They demonstrate just how dramatic this set of optimizations can be when applicable. For each loop and query we see the CPU time before and after the optimization, and the number of solver nodes before and after the optimization, with factors of improvement. Note that only source-to-source transformations were used for the optimization: the *test*/1 predicate was simulated using separate rules involving cuts and failure, as described above.

		Time (sec)			Solver Space		
Loop	Query	Before	After	(Factor)	Before	After	(Factor)
Tree	Search	5.71	1.11	(5)	52,622	2142	(25)
Simulation (UV)		0.38	0.08	(5)	3564	246	(14)
Simulation (2 vars)		0.75	0.14	(5)	7044	408	(17)
Simulation (5 vars)		1.11	0.18	(6)	17,568	1778	(10)
Mortgage	1	8.34	0.43	(21)	66,781	1082	(62)
	2	6.89	0.46	(15)	64,617	1429	(45)
Return mortgage		9.28	0.32	(29)	3249	2891	(1)
Greedy		0.36	0.08	(4)	5456	708	(8)
Layout		0.33	0.09	(4)	4997	778	(6)
Invest		7.13	1.10	(6)	77,554	7568	(11)

Tree is a binary search tree lookup, where each node traversed contributes to a cost relative to an initial cost, and the cost is bounded above. *Simulation* is a set of numerical simulations of continuous systems. The state variables are bounded above and increase monotonically. They have one, two, and five state variables, respectively. Those with multivariate state have multiple redundant constraints that must be tested together. *Mortgage* is a compound interest calculation program that has become a popular CLP($\mathscr{R}$) benchmark. The third example is return future redundant. Interestingly, it shows that space saving is not the only source of speedup: actual constraint solving time is saved. *Greedy* is a naive packing algorithm selecting integers greedily from a list such that their sum is below a bound. *Layout* distributes a list of objects on a page with bounded width, starting a new

row when needed. *Invest* is a large and complicated loop with 12 rules and 12 arguments, with a partially future redundant constraint in each of the four recursive rules. For these results, the rules of the program were re-ordered to isolate the direct effects of the optimization. This tends to double or triple the improvements in performance.

8. CONCLUSION

We have introduced two important optimizations removing redundant inequalities at compile time from an important class of procedures. It is clear that, where applicable, these optimizations will cause significant performance improvement. We have also made considerable progress toward expanding their applicability. We expect that the long-term effectiveness of these optimizations depends on further improved understanding of CLP programming methodology and applications, as well as improved techniques for controlling multiple specialization in practical compilers.

ACKNOWLEDGMENTS

The author is grateful to Nathan Loofbourrow, Bill Pippin, and a number of anonymous referees for comments on earlier versions of this paper and to Iván Ordóñez for his help in preparing the largest of the benchmark programs.

Received February 2, 1995; final manuscript received October 8, 1997

REFERENCES

Jaffer, J., and Lassez, J.-L. (1987), Constraint logic programming, *in* "Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany," pp. 111–119.

Jaffer, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C. (1992a), An abstract machine for CLP($\mathscr{R}$), *in* "Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco," pp. 128–139.

Jaffer, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C. (1992b), The CLP($\mathscr{R}$) language and system, *ACM Trans. Programming Lang. Systems* (*TOPLAS*) **14**(3), 399–395.

Jørgensen, N. (1992), "Abstract Interpretation of Constraint Logic Programs," Ph.D. thesis, Roskilde University Center, Denmark.

Jørgensen, N., Marriott, K., and Michaylov, S. (1991), Some global compile-time optimizations for CLP($\mathscr{R}$), *in* "Logic Programming: Proceedings of the 1991 International Symposium, San Diego, CA" (V. Saraswat and K. Ueda, Eds.), pp. 420–434, MIT Press, Cambridge, MA.

Kemp, D. B., Ramamohanarao, K., Balbin, I., and Meenakshi, K. (1989), Propagating constraints in recursive deductive databases, *in* "Proc. 1989 North American Conference on Logic Programming," pp. 981–998.

Kemp, D. B., and Stuckey, P. J. (1993), Analysis based constraint query optimization, *in* "Proceedings of the 1993 International Conference on Logic Programming," pp. 666–682.

Marriott, K., and Sondergaard, H. (1990), Analysis of constraint logic programs, *in* "Proc. of the 1990 North American Conference on Logic Programming, Austin, TX" (S. Debray and M. Hermenegildo, Eds.), pp. 521–540, MIT Press, Cambridge, MA.

Marriott, K. G., and Stuckey, P. J. (1993), The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering, *in* "Proc. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)," Charleston, NC.

McDonald, A. D., Stuckey, P. J., and Yap, R. H. C. (1993), Redundancy of variables in CLP($\mathscr{R}$), *in* "Logic Programming: Proceedings of the 1993 International Symposium," Vancouver, MIT Press, Cambridge, MA.

Michaylov, S. (1992), "Design and Implementation of Practical Constraint Logic Programming Systems," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, [Available as technical report CMU-CS-92-168].

Michaylov, S., and Pippin, B. (1994), Optimizing compilation of linear arithmetic in a class of constraint logic programs, *in* "International Logic Programming Symposium," Ithaca, New York. [Also appears as Technical Report OSU-CISRC-8/94-TR44, Department of Computer and Information Science, The Ohio State University]

Stuckey, P. J., and Sudarshan, S. (1994), Constraint query optimization, *in* "Proceedings of the ACM-SIGMOD Symposium on Principles of Database Systems," pp. 56–67.

Winsborough, W. H. (1989), Path-dependent reachability analysis for multiple specialization, *in* "Logic Programming: Proceedings of the North American Conference, Cleveland, OH," pp. 133–153, MIT Press, Cambridge, MA.